Stacks

15-213: Introduction to Computer Systems
Recitation 5: February 11, 2012

Joon-Sup Han
Section G
Today: Stacks on stacks on stacks

- News
- Stack discipline review
  - Quick review of registers and assembly
  - Stack frames
  - Function calls (IA32)
  - x86 (IA32) and x86-64
- Example
News

- bomblab is due tomorrow night
  - Don’t use your late days yet
  - But “if you wait till the last minute, it only takes a minute!”
- buflab is coming out tomorrow night
  - All about stacks
- Pro-tip: we love stack questions on exams
  - Students find stack questions difficult
  - Make sure you know stacks well
Quick review of registers (IA32)

- **Caller saved: %eax, %ecx, %edx**
  - You must save these before a function call if you need them

- **Callee saved: %ebx, %edi, %esi**
  - You must save these before any work if you need them

- **Base pointer: %ebp**
  - Points to the “bottom” of a stack frame

- **Stack pointer: %esp**
  - Points to the “top” of a stack frame

- **Instruction pointer: %eip**
  - Generally don’t need to worry about this one
IA32 stack

- This is a memory region that grows down
- Confusingly, refer to the bottom of the stack as the “top”
- %esp refers to the lowest stack address
pushing and popping

- It may be helpful to remember this correspondence (IA32)
  - Note: This is probably not how it actually works

```
pushl src ➔ subl $4,%esp
    movl src,(%esp)

popl dest ➔ movl (%esp),dest
    addl $4,%esp
```

- %esp “points” to the top value on the stack
## Quick example

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>0x110</td>
<td>0x110</td>
<td>0x110</td>
</tr>
<tr>
<td>0x10c</td>
<td>0x10c</td>
<td>0x10c</td>
<td>0x10c</td>
</tr>
<tr>
<td>0x108</td>
<td>0x123</td>
<td>0x123</td>
<td>0x123</td>
</tr>
<tr>
<td>0x104</td>
<td>0x213</td>
<td>0x213</td>
<td>0x213</td>
</tr>
</tbody>
</table>

### pushl %eax

<table>
<thead>
<tr>
<th>%eax</th>
<th>%eax</th>
<th>%eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x213</td>
<td>0x213</td>
<td>0x213</td>
</tr>
</tbody>
</table>

### popl %edx

<table>
<thead>
<tr>
<th>%edx</th>
<th>%edx</th>
<th>%edx</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x555</td>
<td>0x555</td>
<td>0x213</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%esp</th>
<th>%esp</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x108</td>
<td>0x104</td>
<td>0x108</td>
</tr>
</tbody>
</table>
Stack frames

- Every function call is given a stack frame
- What does a C function need?
  - Local variables (scalars, arrays, structs)
    - Space for scalars if the compiler couldn’t allocate enough registers
  - Space to save callee saved registers
  - Space to put computations/temporaries
  - A way to give arguments and call other functions
  - A way to grab arguments
- We can do all these things with the stack!
Function calls

■ Use the stack for function calls

■ Function call
  ▪ call label  Push “return address” on stack, jump to label

■ Return address?
  ▪ Address of the instruction immediately after the call
  ▪ Example from disassembly:
    ▪ 804854e: e8 3d 06 00 00  call 8048b90 <main>
    ▪ 8048553: 50 pushl %eax
  ▪ Return address is 0x8048553

■ Returning from a function call
  ▪ ret  Pop return address [(%esp)] into %eip, keep running
  ▪ Remember that the function’s actual return value must be in %eax
What does this look like?

- `0x804854e: e8 3d 06 00 00` call `0x8048b90 <main>`
- `0x8048553: 50` `pushl %eax`

```assembly
804854e:
   e8 3d 06 00 00
   call 8048b90 <main>

8048553:
   50
   pushl %eax
```

---

![Diagram](image.png)
Returning

- `0x8048591: c3`

```
8048591:
  c3
  ret
```

```
0x110 0x10c 0x108 0x123 0x104 0x8048553
%eip
%esp
0x104
```

```
0x110 0x10c 0x108 0x123 0x104 0x8048553
%eip
%esp
```

%esp
Suppose you have

```c
int scalar_saxpy(int a, int x, int y)
{
    return a * x + y;
}
int main(void)
{
    int x = 3, y = 4, a = 5;
    return scalar_saxpy(a, x, y);
}
```

In IA32 we pass arguments using the stack.

- `scalar_saxpy` grabs arguments by reaching up the caller's stack frame!
What exactly does this look like?

- **Hint:** You have probably seen this in bomblab

```c
int scalar_saxpy(int a, int x, int y)
{
    return a * x + y;
}
```

```assembly
08048374 <scalar_saxpy>:
8048374:   55     push   %ebp
8048375:   89 e5     mov    %esp,%ebp
8048377:   8b 45 08     mov    0x8(%ebp),%eax       # Move a to %eax
804837a:   0f af 45 0c     imul   0xc(%ebp),%eax     # Multiply x with a
804837e:   03 45 10     add    0x10(%ebp),%eax     # Add y to a*x
8048381:   5d     pop    %ebp
8048382:   c3     ret
```
Stacks/Functions on x86-64

- Everything is even easier
- Arguments (≤ 6) are passed in registers
  - %rdi, %rsi, %rdx, %rcx, %r8, %r9
  - Extra arguments are still passed on stack – IA32 knowledge is still useful!
- We have nicer compilers now – don’t need %rbp
- Overall less stack usage - potentially better performance
  - See memory hierarchy
- You are expected to know how 64-bit stacks work
  - Even if there are no labs on it
Example (from a 213 exam I took)

- Given the following function prototypes, and initial lines of IA32 assembly for each function, fill in the stack frame diagram with

  - registers to be saved
  - any arguments to the function foo
  - the return address
  - Any registers stored on the stack by the asm fragment
  - The location on the stack pointed to by %esp and %ebp after execution of the sub instruction

```assembly
int foo(int a, int b, int c, int d);
push %ebp
mov %esp,%ebp
push %ebx
sub $0x10,%esp
```

- (Bonus: How does foo access its arguments after the sub?)
But what about floating point arguments?

- It’s complicated
- You don’t need to worry about it
- In some chips there is a separate floating point stack (!!)
- Example of complication
  - In x86-64 the stack on function entry needs to be 16-byte aligned if a function will need to use floating point (??)
  - Many, many more tricky things to know
Questions?
(stacks, bomblab, what is buflab)

(come to office hours if you need help)